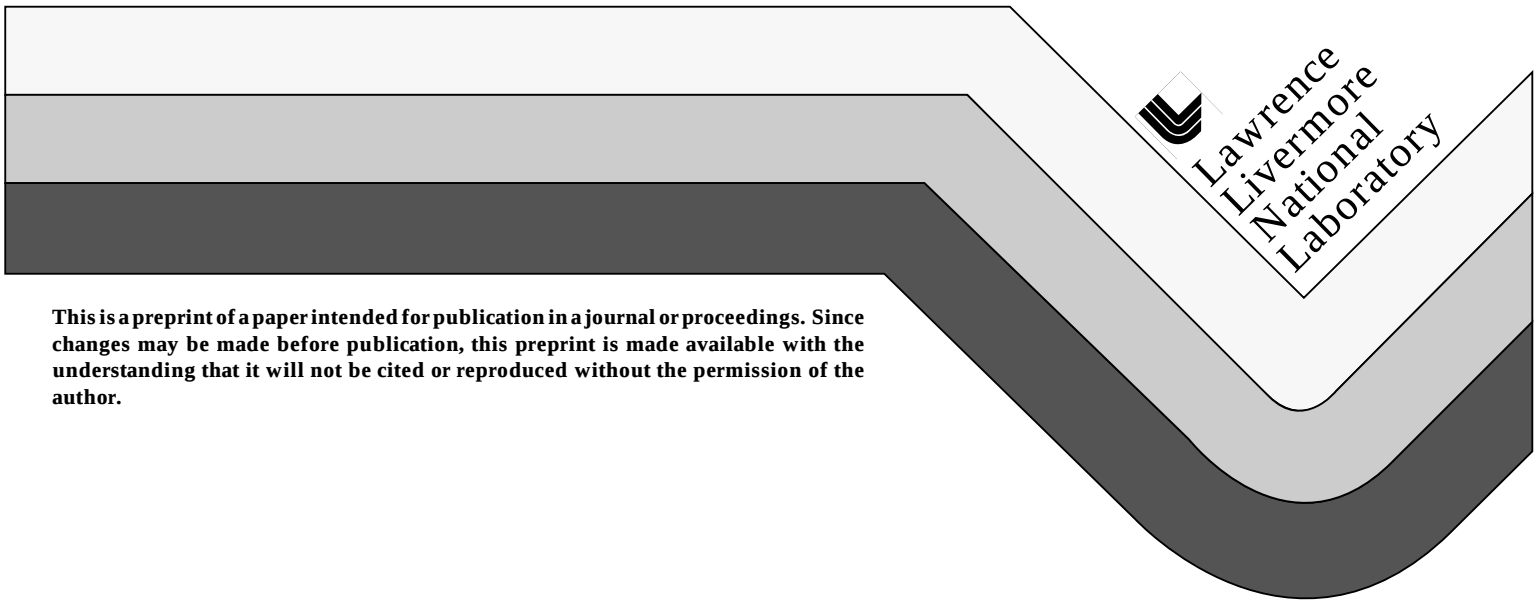
UCRL-JC-119772 Rev. 1
PREPRINT

The Parallel I/O Architecture of the High Performance Storage System (HPSS)

R.W. Watson
R.A. Coyne

This paper was prepared for submittal to the
14th IEEE Symposium on Mass Storage Systems
Monterey, CA
September 11-14, 1995

April 1995



The Parallel I/O Architecture of the High Performance Storage System (HPSS)

Richard W. Watson
Lawrence Livermore National Laboratory
Livermore, CA

Robert A. Coyne
IBM Government Systems
Houston, TX

Abstract

Datasets up to terabyte size and petabyte capacities have created a serious imbalance between I/O and storage system performance and system functionality. One promising approach is the use of parallel data transfer techniques for client access to storage, peripheral-to-peripheral transfers, and remote file transfers. This paper describes the parallel I/O architecture and mechanisms, Parallel Transport Protocol (PTP), parallel FTP, and parallel client Application Programming Interface (API) used by the High Performance Storage System (HPSS). Parallel storage integration issues with a local parallel file system are also discussed.

Introduction

Rapid improvements in computational science, processing capability, main memory sizes, data collection devices, multimedia capabilities, and integration of enterprise data are producing very large datasets, ranging from tens of gigabytes up to terabytes. Both distributed and single-site storage systems must soon manage total capacities scalable into the petabyte range. We expect these large datasets and capacities to be common in high-performance and large-scale national information infrastructure scientific and commercial environments. One result of this rapid growth is a serious imbalance between I/O and storage system performance and system functionality relative to application requirements and the capabilities of other system components.

To correct this imbalance, the performance and capacity of large-scale storage systems available in the general or mass marketplace today must be improved by at least two orders of magnitude with corresponding improvements in architecture and functionality. One promising approach is the use of parallel I/0 [2,4,7,8,14,15,17,20,23,41]. Restoring I/O performance balance is the goal of the HPSS, which is the major development project within the National Storage Laboratory (NSL).

The NSL was established to investigate, demonstrate, and commercialize new mass storage system architectures to meet the needs above [11,12,48]. The NSL and closely related projects involve more than 20 participating organizations from industry, Department of Energy (DOE), and other federal laboratories, universities, and National Science Foundation (NSF) supercomputer centers. The current HPSS development team consists of IBM Government Systems, four DOE laboratories (Lawrence Livermore, Los Alamos, Oak Ridge, and Sandia), Cornell University, and NASA Langley and Lewis Research Centers. Ampex, IBM, Maximum Strategy Inc., Network Systems Corp., PsiTech, Sony Precision Graphics, Storage Technology, and Zitel have supplied hardware in support of HPSS development and demonstration. Intel, IBM, and Meiko are cooperating in the development of high-performance access for supercomputers and MPP clients. The Cornell Theory Center and the Maui High Performance Computer Center are providing integration testbeds for independent verification of HPSS scalability and parallel data access.

Architectural overview

The network-centered HPSS architecture, based on the IEEE Mass Storage Reference Model: version 5 [10,25], has a high-speed network for data transfer and a logically separate network for control (Figure 1) [6,11,24,30,37]. The control network uses the Open Software Foundation's (OSF) Distributed Computing Environment (DCE) Remote Procedure Call technology [39]. In actual implementation, the control and data transfer networks may be physically separate or shared.

An important feature of HPSS is its support for both parallel and sequential I/O and standard interfaces for communication between processors (parallel or otherwise) and storage peripherals. In typical use, clients direct a request for data to an HPSS server. The server directs network-attached storage peripherals or servers to transfer data directly, sequentially or in parallel, to the client node(s) through the high-speed,

data-transfer network (Figure 1). HPSS also supports devices attached to HPSS servers. TCP/IP sockets and IPI-3 over a High Performance Parallel Interface (HIPPI) are being utilized today; Fibre Channel Standard (FCS) with IPI-3 or SCSI, or Asynchronous Transfer Mode (ATM) will be supported in the future [5,28,47,49]. Through its parallel storage and I/O support by data striping, HPSS will continue to scale upward as additional storage peripheralsand controllers and network connectivity are added.

The HPSS components (Figure 2) are multi-threaded using DCE threads [39] and can be distributed and multiprocessed. Multithreading is also important for serving large numbers of concurrent users. HPSS also uses DCE security and distributed time and directory services. HPSS uses the Transarc Encina software for support of atomic transactions, logging, and system metadata [18,32,47]. The storage peripherals managed by HPSS can be organized into multiple storage hierarchies [3,11]. Storage system management (SSM), built around an ISO-managed object framework, is another important HPSS focus [27,29,34]. HPSS components can be run either on single or distributed server machines or on one or more nodes of a parallel system.

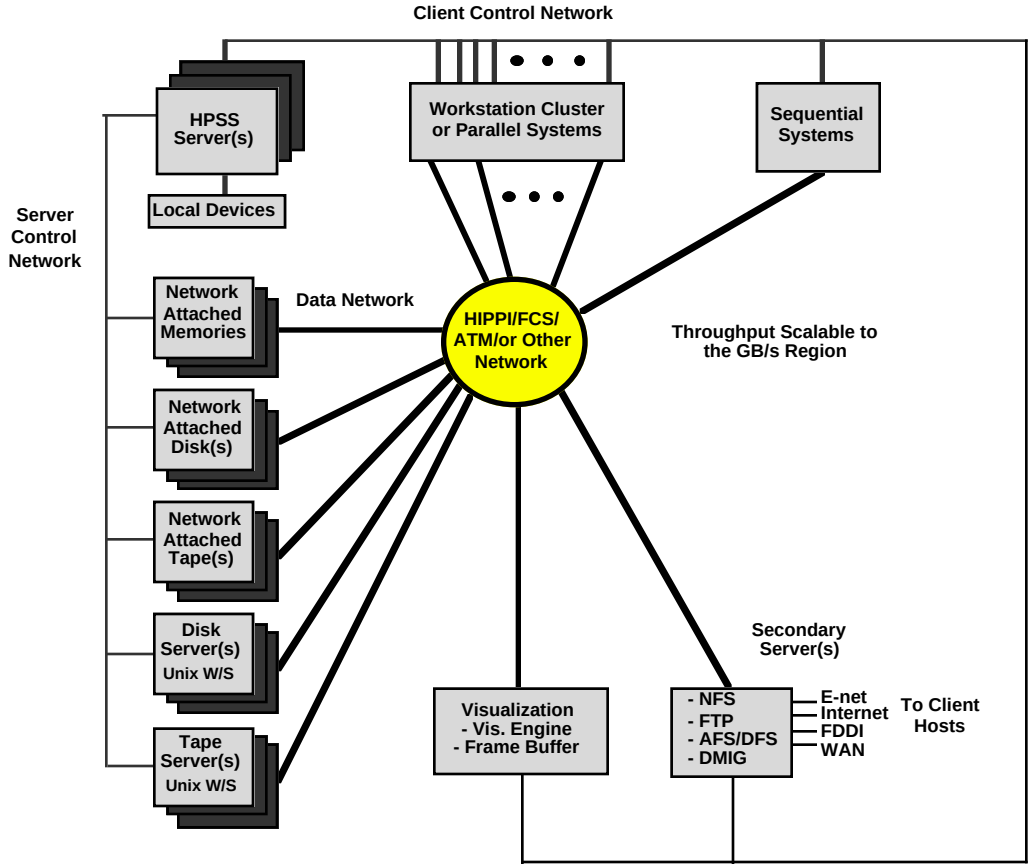


Figure 1. HPSS is designed to support this type of configuration.

The key objectives of HPSS are:

- Scalability in several dimensions, including distribution and multiprocessing of servers, data transfer rates to gigabytes per second, storage capacity to petabytes, file sizes to terabytes, number of naming directories to millions, and hundreds to thousands of simultaneous clients.
- Modularity by building on the IEEE Reference Model architecture (Figure 2) to support client access to all major system subcomponents, replacement of software components during the storage system's lifecycle, and integration of multivendor hardware and software storage components.
- Portability to many vendors' platforms by building on industry standards, such as the OSF DCE, standard communications protocols, C, POSIX, and UNIX with no kernel modifications. HPSS uses commercial products for system infrastructure, and all HPSS component interfaces have been put in the public domain through the

IEEE Storage System Standards Working Group.

- Reliability and recoverability through support for atomic transactions among distributed components, mirroring and logging of system metadata or user data, recovery from failed devices or media, reconnection logic, ability to relocate distributed components, and use of software engineering development practices.
- Client APIs to all major system components, including a parallel Client API and interface to vendor parallel file systems, and support for industry standard services such as File Transfer Protocol (FTP) (sequential and parallel) and Network File System (NFS) [28,43]. Future support will include OFS's Distributed File System (DFS), and the Unix Virtual File System (VFS) [31]. Support is also planned for interface to local and distributed file systems through the Data Management Interface Group (DMIG) standard [13,33].
- Support for better integration with data management systems through appropriate interface functionality at multiple levels in the architecture.
- Security through DCE and POSIX security mechanisms, including authentication, access control lists, file permissions, and security labels.
- System manageability through a managed-object reporting, monitoring, and database framework, and management operations with graphical user interface (GUI) access and control.
- Distributability by building on a client/server architecture and use of an OSF DCE infrastructure.

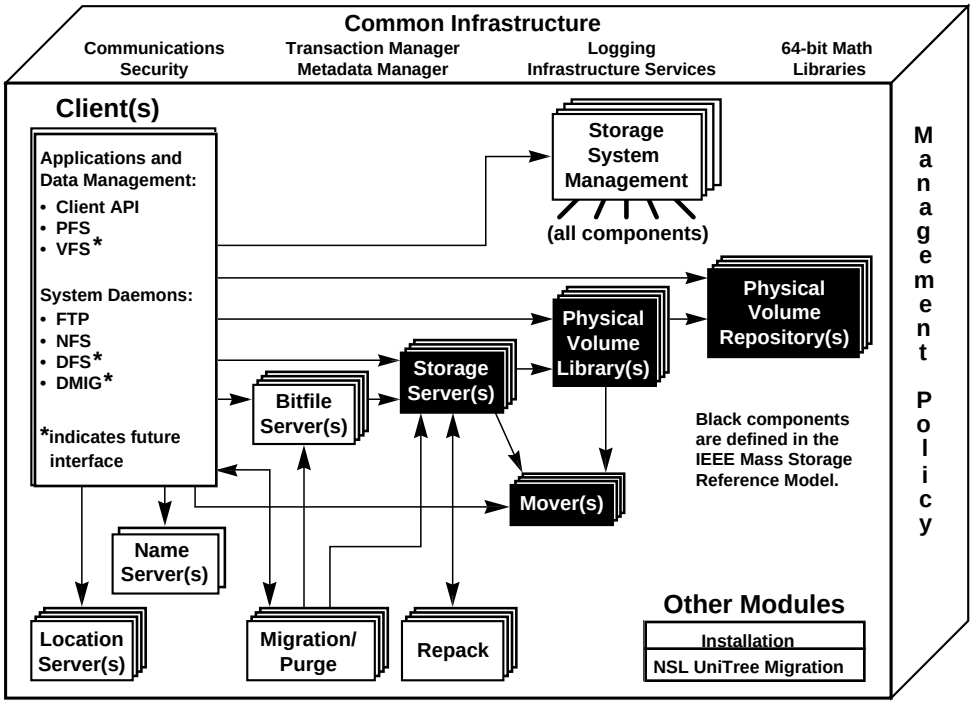


Figure 2. HPSS software model diagram. The components in shaded boxes are defined in the IEEE Mass Storage Reference Model: version 5 [10,25].

The HPSS parallel I/O architecture

Parallel object model

Files are the main high-level object discussed in this paper but, because all HPSS component APIs are accessible, other types of objects, such as databases, can be built on lower-level HPSS abstractions such as logical segments, virtual volumes, or physical volumes [22,44]. All objects have unique object identifiers. Human-oriented naming can be provided by separate naming servers. An HPSS parallel file is logically a sequence of up to 2^{64} bytes in length. Data is addressed by byte offset and length. The offset is specified either through the POSIX file pointer

mechanism or by explicit offsets. A general scatter/gather access is supported. Support is also provided for explicit staging of whole and partial files between levels of the storage hierarchy. The stripe width and block size parameters specify how the parallel file is laid out on logical volumes (Figure 3).

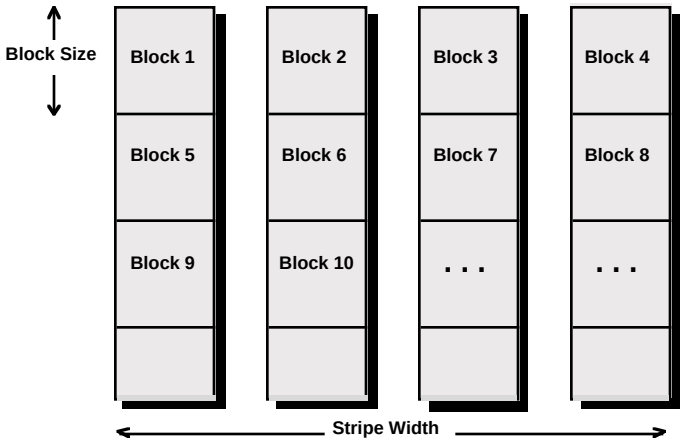


Figure 3. A file striped four ways on four virtual volumes.

The stripe widths and block sizes available at each level within the many possible HPSS storage hierarchies [3] are established by the system administrator through the GUI SSM interface [34]. For example, a given set of disks might be grouped in striped virtual volumes of widths 1, 2, 4, and 8. A set of tapes at the next level of the hierarchy might only be available in striped virtual volumes of widths 1, 2, and 4. This means that for migration both the 4- and 8-way striped disk files would migrate to 4-way-striped virtual tape volumes in their respective storage hierarchies but would cache back to disks as 4- or 8-way stripes as appropriate for their specified hierarchy. HPSS supports any stripe width, not just powers of two, although the latter is expected to be common. Setting the relative stripe widths and block sizes at different levels of a given hierarchy to multiples of each other, respectively, will generally lead to increased parallel I/O and higher throughput.

When files are created, the storage hierarchy, stripe width, and block size are specified either implicitly or explicitly by class-of-service (COS) parameters within the HPSS open/create function [35]. The application might implicitly specify latency and bandwidth, which are then mapped to an appropriate storage hierarchy and parallelism by the bitfile server (BFS) with mapping data provided by the storage server (SS). The appropriate mappings between the COS parameters and actual stripe width, block size, and hierarchy specifications are set when virtual volumes are created by the system administrator through the HPSS SSM interface. Alternatively, the application can explicitly specify the hierarchy, stripe width, and block size; however, these must conform to specifications made available by the system administrator.

The main design objectives and central concepts in the HPSS parallel I/O architecture are given in the PTP section below.

Parallel transport protocol

During the past two years, the PTP has influenced, and been influenced by, the industry-government-university collaboration working toward a protocol that supports the parallel data exchange of datasets that meet the objectives below. The PTP and the HPSS experience are also being used by the IEEE Storage System Standards Working Group P1244.MVR standards group [26]. The PTP and HPSS parallel I/O design goals are to [1]:

1. Provide parallel data exchange between heterogeneous systems and devices. Sources and sinks can be any distributed combination of storage abstractions such as segments of files, memory buffers in heterogeneous systems, physical peripherals, or network addresses.
2. Support different combinations of parallel and sequential sources or sinks.
3. Support network-attached peripherals.
4. Support gather/scatter and random access across heterogeneous systems. In particular, any combination of stripe widths, blocking factors, and regular or irregular data blocks can exist on the source and sink sides of the transfer (i.e., no assumptions can be made about source and sink data layouts).
5. Provide I/O bandwidth improvements that implicitly scale with increasing physical parallel connectivity.

6. Maintain independence from the lower level transport protocol. The PTP should work with any type of network and reliable transport protocol (e.g., TCP/IP, HIPPI, FCS, ATM).
7. Maintain independence of data paths. Data flowing on any parallel path in a given parallel transfer should flow asynchronously to that on any other path.
8. Support efficient sequential I/O as a special case.
9. Support the separation of data and control needed for a flexible parallel I/O architecture.

The PTP sits above the Transport layer in a network architecture and can be used by a wide variety of higher level services and APIs. It is beyond the scope of this paper to give the PTP in detail but a brief description follows [1]. The general PTP configuration and network model is shown in Figure 4.

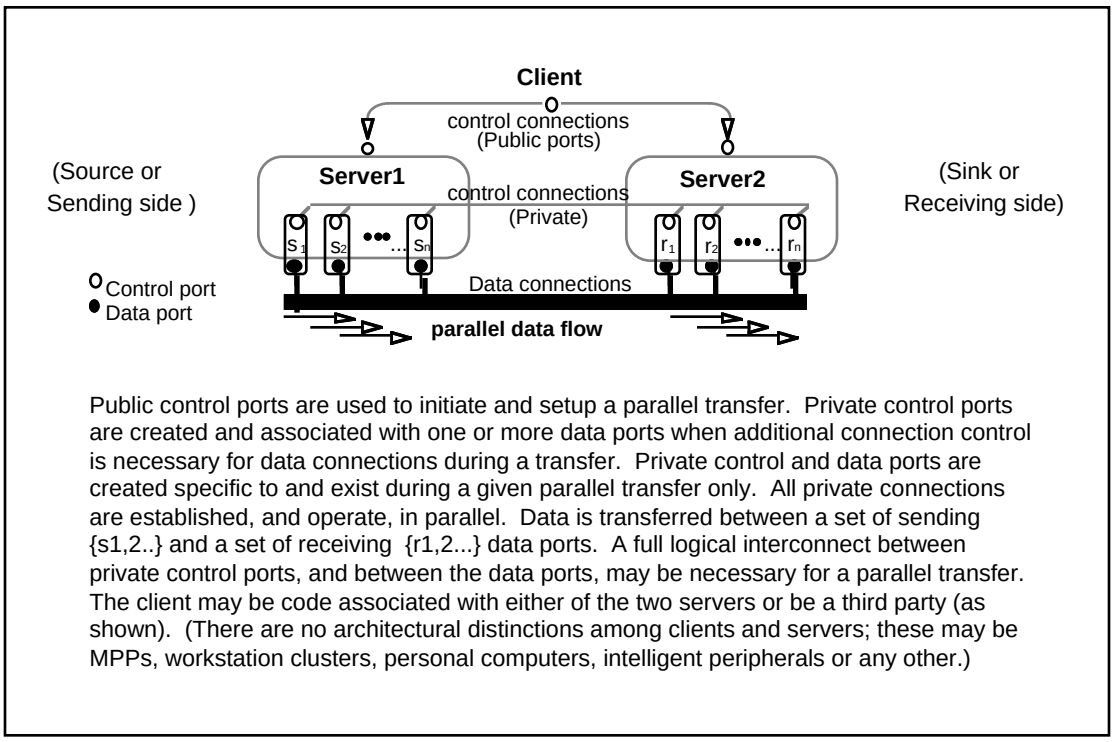


Figure 4. Parallel transport model.

The PTP model assumes three entities, the client (that logical entity initiating a transfer), a source server (Server 1, the source of data), and a sink server (Server 2, the sink of data). The client could be a third party or a module collocated at either the source or sink servers. The model definition assumes reliable connection mode transport communication services. A connection-oriented structure is asymmetric; the passive side listens for connection requests and the active side initiates a connection.

Two classes of connections, control and data, are defined as logically separate, although they may or may not share the same physical network(s). Control connections are characterized by small packets of requests and replies and data connections by large packets of data. A set of data connections are used for parallel transport.

In general, the source and sink data may be distributed over any of the abstractions listed in the first design goal above and be located anywhere in the network. Thus, the source and sink servers shown in Figure 4 may be recursively made up of many distributed cooperating entities.

The PTP defines: (1) the data structures that specify the distributed source data (gather-list) and the distributed sink data (scatter-list); (2) the logical mapping mechanism between the gather/scatter lists; (3) the control information that must be exchanged among client, source, and sink to control the transfer; and (4) the mechanism to specify the detailed transfer plan of required connections and which data flows over which connection.

The data structure that specifies the gather/scatter-lists in requests exchanged among clients and servers is called an I/O descriptor (IOD). The corresponding data structure used in replies is called an I/O reply (IOR). In an actual implementa-

tion, such as in HPSS, a server at one level can be a client of another server at a lower level.

The IOD data structure contains the following information:

- Unique transfer ID generated by the Client API code.
- Function (read or write).
- Source-descriptor-list (gather-list of data source locations).
- Sink-descriptor-list (scatter-list of data sink locations).

The IOR record contains the following information:

- Unique transfer ID generated by the Client API code.
- Set of flags indicating successful or unsuccessful completion and an error code.
- Source-reply-list (description specifying source data results).
- Sink-reply-list (description specifying sink data results).

The concept introduced to support the mappings from the data locations specified in the gather-list of source-descriptors to the data locations specified in the scatter-list of sink-descriptors is called the transfer window (or window). The window is viewed as containing the *n* logically contiguous bytes of the total transfer. Each source-descriptor in the gather-list specifies an offset and length of *m* unique bytes (where *m* may be different for each descriptor) to tell where the source data is to be placed in the window. Each sink-descriptor in the scatter-list specifies an offset and length of *k* unique bytes (where *k* may be different for each descriptor) to tell where the sink data is to be obtained from the window. The source/sink descriptors also contain an address structure that defines the location of the actual data. This address structure can take several forms (e.g., network address/port; file ID, offset and length; or stripe specification). The stripe specification contains an address (e.g., peripheral, memory buffer, port) and an offset that determines where to begin to get or put data from/to the peripheral. There is also a block size and a stride (i.e., the number of blocks to skip between blocks) defining how to get or put data from/to the window to/from the peripherals. Figure 5 illustrates the window concept.

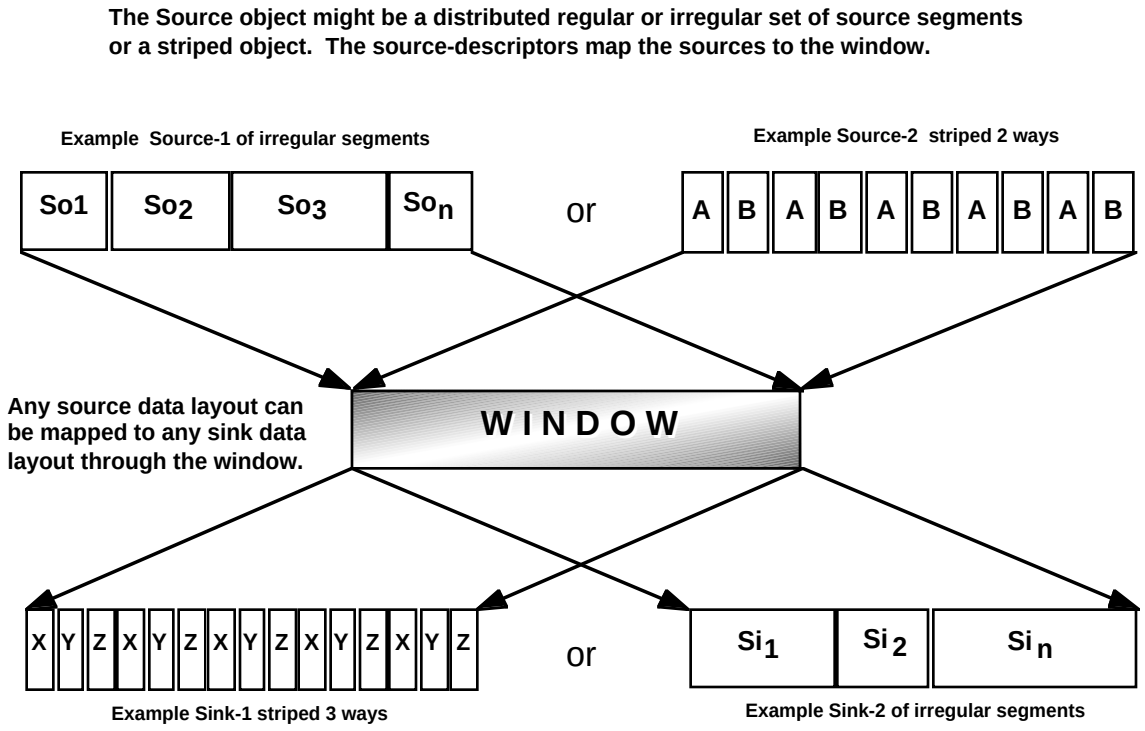


Figure 5. Sample mappings between gather-lists and scatter-lists using a logical window show how different source structures can be mapped to different sink structures.

A user or application requests the PTP client to initiate a transfer. The PTP client might be in a parallel I/O library or a file transfer client module, as shown in later examples. Well known network control addresses (control ports) are assumed to be publicly available for the PTP client and source and sink servers. Public control connections are sequential setup exchanges used to initiate, control, and describe the transfer. Private control connections may be established during the transfer as necessary to control transfers on specific data connections. Requests and replies exchanged over these control connections can be performed in parallel. Data is moved in parallel over sets of data connections between storage objects in the source to storage objects in the sink. Data connections are private (i.e., created transfer specific).

The basic mechanics underlying PTP are quite simple:

1. The client connects to the well known control port of the passive server and sends a unique transfer ID and a message to move data. This message contains the IOD described earlier to specify the function (read or write), gather/scatter lists, and window mappings. Either the source or sink can initially be the passive server. If the client is collocated with one of the servers, that server is initially made passive to save control exchanges. The passive server replies to this request with an IOR containing a set of descriptors with ports (network addresses) to use for the transfer and their window relationship to the data described in the IOD. These ports, depending on the lower level protocol in use, are either control ports to be used to establish and control the data connections, or the actual data ports to use for the transfer. All passive ports then listen for connect requests from the active server.
2. The client connects to the well known control port of the active server and sends the unique transfer ID in an IOD structure that includes the passive server's source or sink descriptors mapped to the ports returned in the IOR above.
3. Private control ports set up data ports (or data ports have been defined directly). The active data ports connect in parallel to the listening passive data ports and transfer data in parallel. For a detailed example of how data is transferred on a single data connection see Reference [24]. During control exchanges between Movers (MVRs) at the start of the data transfer phase, the passive/active roles can be renegotiated, as appropriate to the types of peripherals involved.
4. The servers reply to the client with completion status in IORs.

Each server manages the movement of data between its peripherals and data ports. The protocol is independent of how data is distributed within the server. (This distribution is specific to both a given system and the hardware involved.)

A central mechanism of the HPSS parallel I/O architecture determines the data transfer plan of the PTP by using successive mappings of that part of the IOD structure for which HPSS is responsible. The Client API code provides the necessary detailed information in the IOD for that part of the I/O for which it is responsible. The HPSS part of the IOD is expanded as it is passed through HPSS's layered BFS, SS, and MVR abstractions. Each HPSS server only has to know as much about the parallel I/O as is necessary for its role. The data transfer plan and mappings to and from the transfer window, outlined above, are thus developed as the IOD(s) travels through the HPSS servers. Ultimately, at the MVR level, IODs contain explicit communication ports, peripheral addresses, offsets and lengths and their relationships to window offsets and lengths for the complete PTP parallel data transfer.

Role of the HPSS components in parallel I/O

Application interfaces

HPSS supports several high-level interfaces: Client API, standard and parallel FTP, and NFS. DFS and VFS are planned for future releases. We discuss the Client API, FTP, and NFS below, and illustrate their support of parallel I/O and parallel files in the examples. The DMIG specification for integration with other file systems [13,33] will also be supported, although it is not a Client API. All the client interfaces contain both control code to form IODs and communicate with HPSS and client Mover code to perform the parallel I/O. Details of how the Client API code interfaces with a given parallel system are system dependent although the main modules are portable.

Client API. The HPSS client-to-file server API mirrors the POSIX file system interface specification with enhancements [45]. Client API supports extensions that allow the programmer to take advantage of specific features provided by HPSS (e.g., COS passed at file creation and support for parallel data transfers). The HPSS Client API supports client application access to HPSS files and integration with other services such as FTP, NFS, or a local parallel file system. The Client API code contains the client and passive server defined in the PTP, with HPSS playing the active server role. While the base Client API code is portable, a system-dependent module is

required to map machine IDs and buffer addresses to network port IDs.

In parallel I/O, the key API functions are the:

- HPSS open/create function that determines the storage hierarchy, stripe width, and block size of the file
- Standard POSIX HPSS read and write functions that implicitly support reading or writing a parallel file into or out of a single buffer or buffer list
- HPSS readlist and writelist functions that support explicit specification of multiple sinks and/or sources of data using the IOD data structure.

The HPSS read, write, readlist, and writelist client functions are mapped by the Client API code into BFS reads and writes that take the IOD as an argument and return the IOR structure. The Client API code also contains MVRs (see below) to manage its half of the data transfer phase of the PTP. Asynchronous I/O, data caching/buffering, and collective primitives (in which many processes perform I/O as a group) can be provided by higher level functions, if desired [4,42]. Other higher level APIs can be constructed using the Client API code. For example, one being implemented to access HPSS from Meiko CS 2 and Cray T3D systems, also portable to other systems, is the MPI-IO API [7].

FTP (standard and parallel). HPSS provides a standard FTP server interface to transfer files between HPSS and a local file system. In addition, parallel FTP (PFTP) an extension and superset of standard FTP, provides high-performance data transfers between client systems and HPSS via parallel data paths.

NFS. The NFS V2 server interface for HPSS provides transparent access to HPSS name space objects. NFS only supports non-parallel transfers to the clients. The goal of HPSS NFS server design and implementation is to provide the same throughput and NFS transaction rate as the native operating system NFS on which the HPSS NFS server runs. Client systems to both the native HPSS and the NFS V2 service see the same name space and data. The NFS V2 server caches data from HPSS in large blocks at high speed using third-party, parallel data paths.

Parallel file system. HPSS can act as an external hierarchical file system to vendor parallel file systems (PFS). The first PFS/HPSS integration, discussed later, supports the IBM SPx parallel I/O file system (PIOFS). Early deployment is also planned for Intel Paragon, Meiko CS-2, and IBM RS 6000 cluster PFS integrations with HPSS.

Name server

The name server (NS) maps a user POSIX path name to an HPSS object. The NS provides a POSIX view of the name space, which is a hierarchical structure consisting of directories, files, and links. Namable objects can be any object identified by HPSS Storage Object IDs, such as sequential or parallel files or parallel virtual volumes. No knowledge of parallel I/O is required of NS. The NS in cooperation with the BFS also provides the POSIX file protection services.

Bitfile server

The BFS provides the POSIX byte stream file abstraction to its clients, supporting bitfile sizes up to 2^{64} bytes. Bitfiles can be striped as illustrated in Figure 3. The BFS supports both sequential and parallel read and write of data from/to bitfiles. Sequential I/O is an efficient special case of parallel I/O with stripe width of 1. Parallel files with a given stripe width and block size are created during the HPSS create call described earlier.

The reads and writes to a file can be random, with the system managing reads and writes of partial files. There are two flavors of reads and writes. The first is a standard POSIX read or write with the parallelism and blocking into or out of the specified buffer(s) handled implicitly by the system. The second is a form of read and write, called readlist and writelist, wherein the application, using IODs, can explicitly define the data layout of sources or sinks and mappings between these layouts and a striped HPSS file. Depending on how HPSS is integrated with a given environment, an application buffer layout could be on different nodes of a SMP or MPP, on different systems in a distributed set of workstations or other systems, or on a workstation cluster. The data layout can be regular or irregular contiguous chunks or be striped across nodes. Similarly, the IOD can specify arbitrary layouts on an HPSS file.

The Client API parallel I/O support code (library) provides for control messages to and from the NS for mapping file path names to bitfile-IDs and to and from the BFS for the BFS_Read, BFS_Writes and other functions. The two control parameters exchanged between the Client API library and the BFS are the IOD and the IOR defined earlier. We use only the IOD in the examples; IORs are assumed.

The BFS deals only with the storage segment abstraction provided by the SS and is itself largely unaware of the parallel I/O mechanisms. The BFS takes the bitfile IDs, offsets and lengths, and window offsets and lengths that it receives in IODs and maps them to one or more new source/sink descriptors containing storage segment IDs, offsets and lengths, and corresponding window offsets and lengths. The

BFS then passes the expanded IOD to the SS. The BFS may also interact with the SS to create or destroy storage segments when necessary.

Storage server

The SS provides a hierarchy of storage object abstractions: logical storage segments, virtual volumes, and physical volumes. All three layers of the SS can be accessed by appropriately privileged clients. For example, a parallel DBMS could be implemented directly on the SS storage segment layer. In this paper, we use the bitfile abstraction for all examples.

The SS translates references to storage segments into references to the corresponding virtual volumes and finally into physical volume references and peripheral addresses. It also schedules the mounting and dismounting of media through the PVL. An important simplifying SS design assumption is that storage segments do not span virtual volumes. The SS, in conjunction with the MVR, has the main responsibility for orchestration of HPSS's parallel I/O operations. The SS is responsible for all storage segment allocations on virtual and physical volumes.

The SS receives storage segment IDs, offsets and lengths in IODs from the BFS, and maps them first to the appropriate virtual volume IDs, offsets and lengths, then to physical volume IDs and offsets and lengths, specifying during this process appropriate window offsets and lengths. The SS then issues an atomic mount to the PVL [16] component to mount the physical volumes in the virtual volume and obtain the appropriate MVR ID. In the case of disk volumes, this is done only once when the volume is first encountered to check accessibility and the associated MVR ID is cached for future references. Disk volumes are also sharable for concurrent accesses. Atomic mounts for removable media are used as part of the deadlock avoidance mechanism [16]. After the mount is completed, the SS forks off multiple parallel threads to handle the I/O for each physical volume and associated I/O peripheral. The SS threads then pass on the expanded IOD structures with peripheral address information to the MVRs associated with the specified peripherals to carry out the data transfer specified by the IODs.

Physical volume library

The PVL manages all HPSS physical volumes. Clients, such as the SS, can ask the PVL to mount and dismount sets of physical volumes and query the status and characteristics of physical volumes. (If the physical volumes are disks, mount/dismount operations are essentially no-ops, as in the SS description above). The PVL maintains physical volume-to-cartridge and cartridge-to-PVR mappings and PVR location. The PVL also controls all allocation of drives. The PVL does not understand the concept of parallel I/O and only executes the requested atomic or non-atomic physical volume mounts. All volume mount requests from all clients are handled by the PVL. This allows the PVL to prevent multiple clients from deadlocking when trying to mount intersecting sets of volumes [16]. The standard HPSS mount interface is asynchronous (i.e., multiple mounts can go on concurrently).

Physical volume repository

The physical volume repository (PVR) manages all HPSS-supported robotics devices and their media, such as cartridges. The PVR supplies generic service and support for devices such as Ampex, STK, and IBM robot services and an operator-mounted peripheral service. Clients, such as the PVL, can ask the PVR to mount and dismount cartridges. Every cartridge in HPSS must be managed by exactly one PVR. Clients can also query the status and characteristics of cartridges. The PVRs contain no knowledge of parallel I/O.

Mover

MVRs are responsible for transferring data from source devices to sink devices. A peripheral can be a standard I/O peripheral with geometry (tape, disk), or a peripheral without geometry (network, memory). There are MVRs for each type of peripheral and network. MVRs also perform peripheral control operations and the control and transfer for both sequential and parallel data transfers. The MVRs, along with the SSs, are the entities that are primarily responsible for HPSS parallel I/O. The MVRs perform the data transfer part of the PTP, using the appropriate network transport protocols as determined by source(s) and sink(s) connectivity (e.g., parallel transfers using IPI-3 on HIPPI or Socket TCP/IP connections). The MVRs determine the final data transfer plan of what data in what sequence goes over what required connections using the information in the detailed IODs they receive from the SS threads. There are MVRs on both the client and HPSS sides of a transfer with essentially identical code and functionality to implement their roles in the PTP.

Storage system management

The HPSS SSM architecture is based on the ISO managed object architecture [27,29]. The Storage System Manager monitors and controls the available resources of the HPSS storage system in ways that conform to the particular management policies of a given site [34,45]. With respect to parallel I/O, we have already mentioned the important role of SSM in

providing the system administrator interface to control the configuration of the striping and blocking factors at each level of the storage hierarchies, the establishment of multiple hierarchies, and the grouping of physical volumes such as disks and tapes into striped virtual volumes. SSM also supports the system administrator in establishing the mapping of COS parameters to virtual volume and other I/O characteristics. Setting up the striping and blocking at each level of a hierarchy as multiples of each other illustrates how much care the system administrator must exercise to achieve optimal performance. Managing stripe widths for tape under various load assumptions is also important [21].

Migration and caching

Automatic migration and caching within storage hierarchies are controlled by a migration/purge server using the appropriate BFS and SS. HPSS also supports explicit caching and migration through stage and purge commands in the Client API code. If necessary, purge migrates the specified data before deleting it. The system administrator uses the GUI interface and SSM services to organize the devices managed by HPSS into storage hierarchies of virtual volumes at each level with specified stripe widths and block sizes. The MVRs perform the appropriate blocking or deblocking and data transfers as storage objects such as files are migrated down the hierarchy (e.g., from faster, more expensive devices to slower, less expensive devices) or cached up the hierarchy in the opposite direction (see Example B below). Caching of partial files to the top of the hierarchy is triggered by an access or stage request. When the block containing the requested data arrives, it is delivered to the requester without waiting for the entire cache or stage to complete. File migration is triggered by higher level volumes filling up. The choice of files and parts of files to migrate is based on parameters such as size and access history.

Examples of HPSS parallel I/O

Four examples of parallel I/O follow: direct Client API, peripheral-to-peripheral copy, parallel I/O for caching and migration to/from the NFS server, and PFTP. The first two examples are given in some detail to illustrate the concepts and mechanisms presented earlier. The remaining two give the general sense of parallel I/O control and data flows. Implicit in all examples is the mapping to or from window offsets and lengths that is taking place as the IODs are handled by each server.

Example A. Application client parallel API

As an example of the application client to HPSS parallel data transfer and the PTP mechanism, consider the four-node parallel HPSS read of a disk file diagrammed in Figure 6. Assume the HPSS-to-client integration supports scatter/gather across nodes. The client in the PTP description is part of a parallel application library containing HPSS Client API and PTP source/sink server and MVR code. Thus, for this example, the sink server and associated client MVRs of the PTP description are also within the client code. The PTP source server is HPSS. In somewhat simplified form, the following communications (numbers correspond to message numbers on Figure 6) take place:

1. The application calls HPSS_Open to open the HPSS disk file. The Client API code sends a message to the NS and receives a bitfile-ID in return. The Client API code calls the BFS_open and receives a bitfile-descriptor, which the Client API maps to a POSIX file descriptor and returns this descriptor to the application. The application then issues an HPSS readlist function to the Client API Code with an IOD that in the source-descriptor-list has the file-descriptor, offset and length and the sink-descriptor-list has the machine IDs and buffer addresses of the application machine's four sink nodes.
2. The Client API library code logically communicates with its collocated sink server code. This client-system-dependent code maps machine IDs and memory addresses to port IDs. Using the port IDs returned, the IOD sink-descriptor-list is modified to contain the port IDs. The Client API code then issues a BFS_Read call to HPSS with the expanded IOD. HPSS in this case is the PTP active server.
3. The BFS then modifies the IOD's source-descriptor-list, mapping the file-descriptor, offset, length to corresponding storage segment IDs, offsets, lengths for this operation. An SS_Read with the expanded IOD is then issued.
4 The SS first maps the storage segment IDs, offsets and lengths to virtual volume IDs, offsets and lengths. It then creates physical volume IODs for each physical volume and forks threads, one for each physical volume, to perform the corresponding I/O for an IOD. The threads map the source-descriptors in the IODs to peripheral addresses, offsets and lengths and issue MVR-Reads to the MVRs corresponding to the devices.

5. The MVR Threads , in parallel, using the information in the completely specified IODs each receives and the unique transfer ID (TID), establish connections to the listening application client MVRs. The data transfer protocol on each connection contains a header with a TID, offset and length along with the data so that the data on each connection can be sent asynchronously in parallel and be properly placed in the client buffers [24]. The HPSS/client MVR pairs do the data transfers in parallel.
6. The client MVRs store the data in the client buffers.

Finally, although not shown in Figure 6, IORs containing completion status are returned up their respective chains.

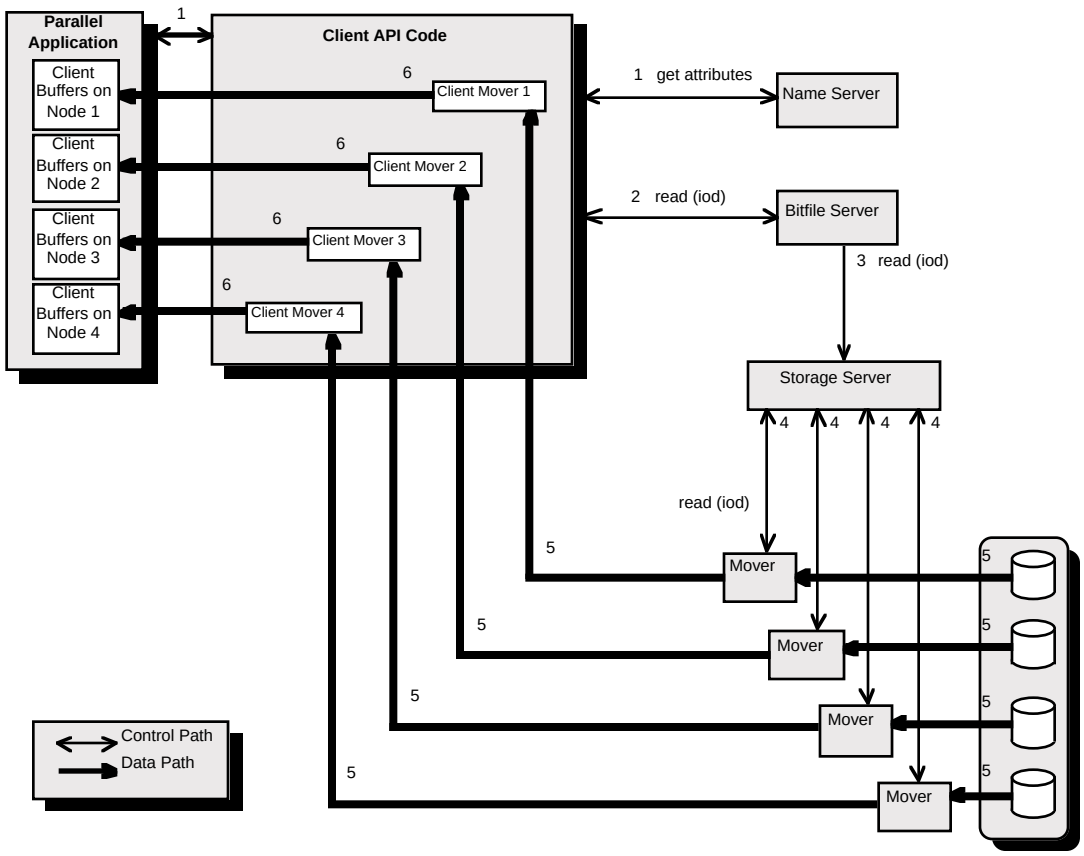


Figure 6. Client parallel transfer.

When a read is from parallel tape, the following additional steps are required. The tape SS sends an atomic mount request to the PVL, which in turn sends mount requests to the PVR. When the atomic mount operation is complete, the PVL notifies the tape MVR to read the tape labels and then reply to the SS. The SS then sends read requests and IODs to the MVRs. The tape MVR then positions the tapes before initiating the data movement.

Example B. HPSS peripheral-to-peripheral parallel copy

This is a more complicated example showing the use of the PTP in a peripheral-to-peripheral, disk-to-tape transfer within HPSS. This particular example is a parallel file copy, but the central data transfer mechanism would also be used during migration and caching of data between devices at different levels of the storage hierarchy. A third-party client initiates the transfer but it is the BFS that plays the role of the PTP client ,and the tape and disk SSs and their associated MVRs play the PTP sink and source servers, respectively.

Figure 7 illustrates the control and data flow for the operation. Within the boxes are the HPSS component subsystems. The multiple MVR Threads are shown but not the multiple SS threads. In the control flow depicted, data is being copied in parallel from disk to tape.

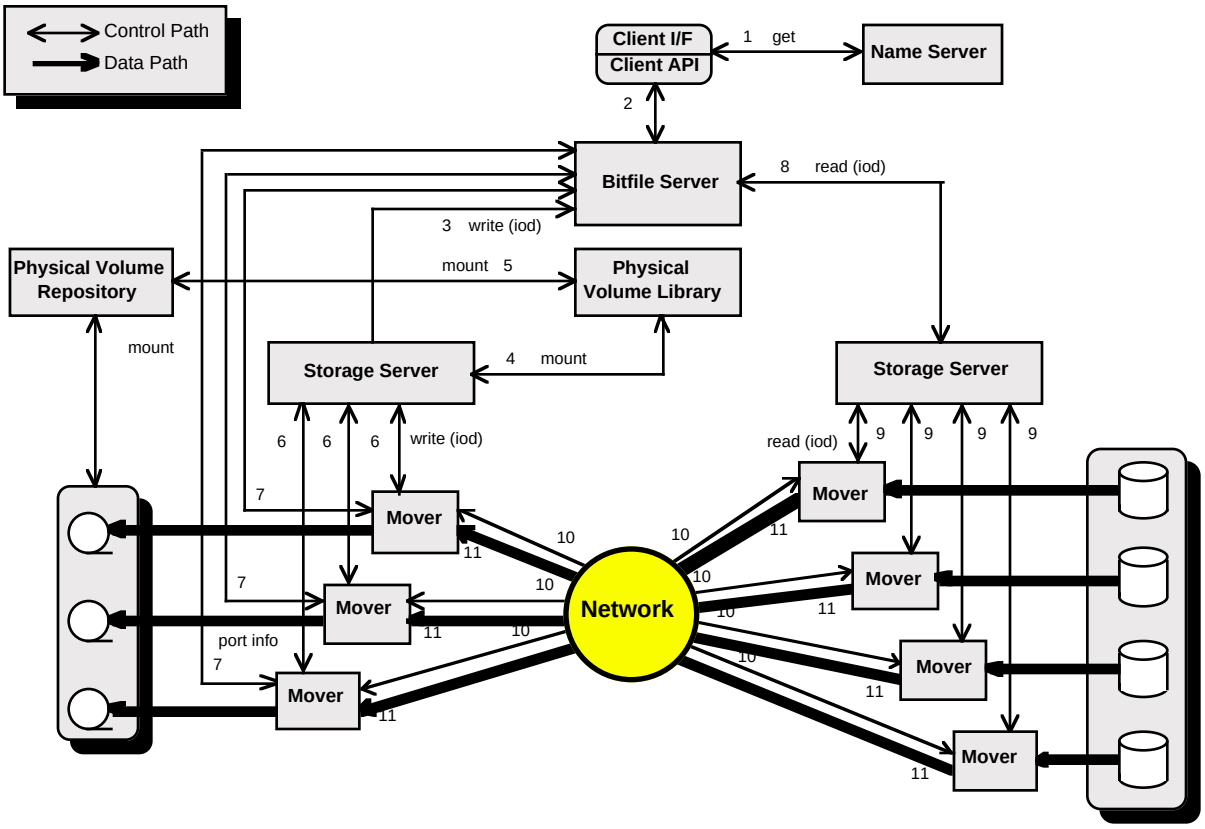


Figure 7. HPSS parallel-to-parallel transfer.

The file on disk is striped across four devices and is being transferred as a striped file to three tape devices. Steps in the parallel transfer are listed below with numbers in parentheses corresponding to messages sent over the numberd pathes in Figures 7.

Control flow for disk file open:

- A client calls the HPSS open API to open the disk file being transferred.
- (2) The open API issues a get attributes request to the NS to retrieve the bitfile ID corresponding to the name of the file being transferred. The external file name is a POSIX file path name.
- The open API issues an open call to the BFS, obtaining an open bitfile-descriptor.
- The open API maps the bitfile descriptor to a POSIX file descriptor.

Control flow for tape file open:

- A client calls the HPSS open API to open (and create) the parallel tape file, supplying appropriate COS parameters. The external name is a POSIX file name.
- (2) The open API issues a create request to the BFS.
- (2) The BFS returns the bitfile ID to the open API.
- The open API issues an open request to the BFS which returns an open bitfile descriptor.
- (1) The open API issues an insert request to the NS to associate the external file name with the bitfile ID returned.
- The open API maps the bitfile descriptor to a POSIX file-descriptor.

Control flow for data movement:

- The client calls the HPSS writelist API to copy the file from disk to tape supplying an IOD with source/sink POSIX file descriptors, offset, length and window offset and length information.
- (2) The IOD above is modified to pass bitfile source/sink information to the BFS in a write request.
- (3) The BFS translates the request into a logical segment and issues a create segment request to the tape SS.
- The tape SS returns a segment ID to the BFS. As additional storage segments are required during the transfer, this and the previous steps are repeated.
- (3) The BFS issues a write request to the tape SS. The IOD is modified to contain

storage segment sink information used by the SS.

- The tape SS translates the storage segment references to virtual volume references and then to the physical volumes associated with the virtual volumes.
- (4) The tape SS issues an atomic mount request to the PVL to atomically mount the tape physical volumes.
- (5) The PVL issues mount requests to the PVR which manages the tape media type.
- The PVL issues a request to the tape MVRs to read the tape labels to assure the correct tapes were mounted.
- The PVL returns the peripheral address of the tape read/write station (sink information) to the tape SS. Then SS threads are forked to handle the physical volume I/O..
- The tape SS locates MVRs associated with the physical devices for the parallel transfer.
- (6) The tape SS issues a write request to each of the tape MVRs. The IOD has been modified to pass peripheral sink information to the tape MVRs.
- (7) Each tape MVR returns its listen-port addresses to the BFS in an IOR.
- (8) The BFS issues a read request to the disk SS with appropriate segment ± Ds. The tape listen-port addresses are passed in the IOD sink information.
- The disk SS translates the storage segment references to virtual volume references, and then to the physical volumes associated with the virtual volumes.
- The disk SS determines which disk MVRs are associated with the parallel transfer.
- (9) The disk SS forks off threads for each disk volume and each issues a read request to a corresponding disk MVR. The IOD has been modified to pass peripheral source information to the disk MVRs.
- (10) The disk MVRs connect to the tape MVR listen-ports, and send transport options and stripe and blocking information.
- (10) The tape MVRs respond with data port addresses and transfer information. During the transfer, the tape MVRs will be the active entities.
- (11) The disk MVRs connect to the tape MVR data ports and transfer data.
- Once all data is transferred, the disk and tape MVRs send IORs to the disk and tape SS threads which, in turn, send IORs to the BFS.
- The BFS replies to the client.

Control flow for disk close

- The client calls the HPSS close API to close the disk file.
- (2) The close API issues a close request to the BFS to close the bitfile.

Control flow for tape close

- The client calls the HPSS close API to close the tape file.
- (2) The close API issues a close request to the BFS to close the bitfile.
- (3) The BFS issues a (storage segment) unmount request to the tape SS.
- (4) The tape SS translates the storage segment references to virtual volume references, and then to the physical volumes associated with the virtual volume. A request is then issued to the PVL to unmount the physical volumes.
- (5) The PVL issues unmount requests to the PVR to actually unmount the tape cartridges.

Example C. NFS transfer

Figure 8 depicts the components of the HPSS NFS V2 server and their interrelationships. The HPSS NFS Server can run on the same or different machines or nodes as the other HPSS components. This particular example is for an NFS read operation. The numbers indicate the main flow sequence. The NFS mount daemon, which runs as a separate process from the NFS server, reviews the export list to determine if the client may mount the directory. If the mount request is honored, the mount daemon returns an NFS handle used in subsequent requests to the NFS server to obtain NFS handles to HPSS file objects.

NFS clients access the NFS server APIs through the reentrant ONC RPC library. When the NFS server is requested to operate on an HPSS file object, the NFS handle that is sent in the request is verified with the export list.

Requests that can time-out and cause the client to reissue the request are saved in a structure. This structure is queried for requests that are likely to cause duplicates. If a request is determined to be a duplicate, the NFS server also sends the reply data to the structure.

The Client API library provides the control path to HPSS. Both HPSS and local control information are cached in structures managed by the header cache and data cache libraries, respectively.

When a bitfile is read or written, the data is read from or written to the NFS data cache. Bitfile data is retrieved from (written to) HPSS when required by the data cache manager, which provides the high-speed parallel data path to HPSS. The parallel I/O is handled as outlined in previous examples.

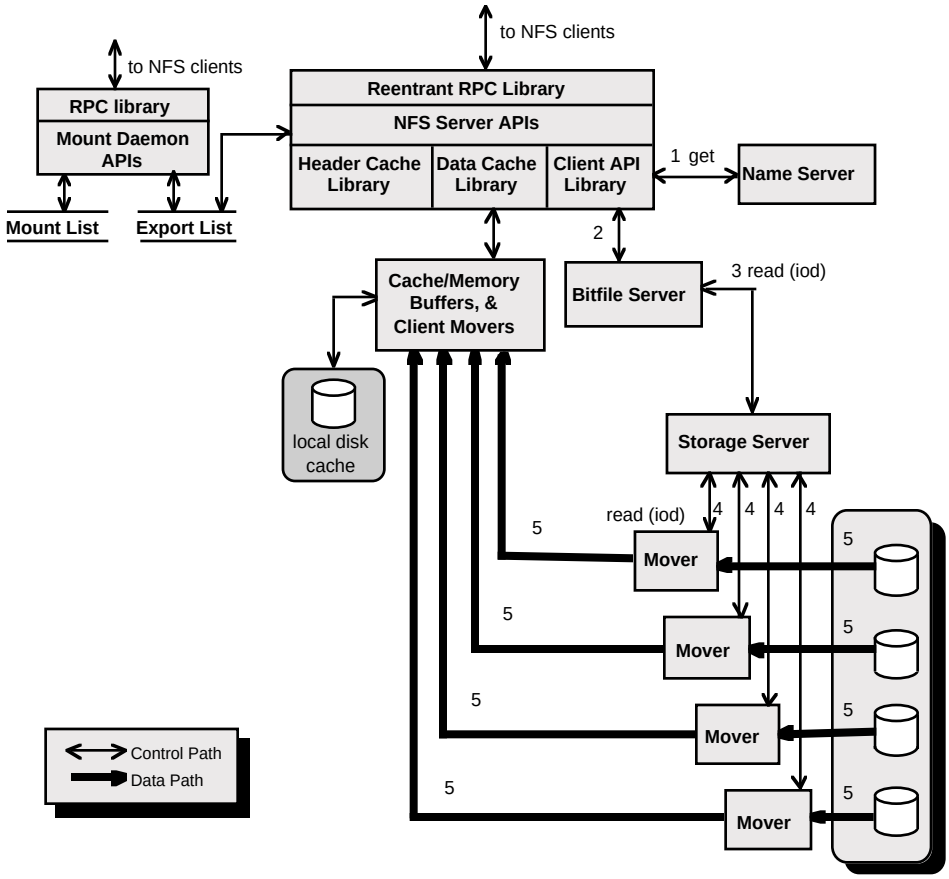


Figure 8. NFS transfer.

Example D. Parallel FTP (PFTP)

The standard Internet FTP [28] supports sequential transfer (get, put, append) of whole files. Its separation of data and control connections supports:

- Control requests and replies on one network or connection and data transfers on another network or connection (e.g., higher speed network).
- A client on one host can initiate a transfer between a source and sink on two other hosts.
- This is exactly the architecture needed for high-speed Third-party FTP service for HPSS.

Extensions to standard FTP are needed to support parallel files and parallel transfers. Those extensions to support parallel I/O are outlined here. Portability is one of the design and implementation goals; to date, the PFTP client has been ported to the IBM RS 6000 supporting TCP/IP socket and IPI-3 over HIPPI data communication, the Intel Paragon supporting TCP/IP socket data communication, and the Meiko CS2 supporting IPI-3 over HIPPI data communication. All implementations use TCP/IP control connections. The daemon extensions also support partial file transfer, which we felt will be important as very large file sizes become common. When we have had more experience with PFTP, we plan to turn the specification over to the Internet Engineering Task Force as a base for possible standardization. At the user command line level, PFTP supports parallel append (Pappend), parallel get (Pget), and parallel put (Pput). During communication between the PFTP client code and the PFTP daemon on the HPSS side, the FTP port, store, and retrieve commands have been extended to support the parallel file I/O and partial file transfer services. Parallel open and close commands have also been added to support partial file transfers.

We started with public domain standard FTP client and daemon code. The FTP client code had to be modified to support multiple MVRs for handling the parallel data transfers and the extended commands. This required a little over 1000 lines of additional C source code to support both parallel TCP/IP and IPI-3 over HIPPI data transfers. The FTP daemon code had to be modified to support the extended commands and to build the IOD for communication with HPSS.

This required about 100 additional lines of C source code.

Figure 9 illustrates the control and data flow for the parallel FTP interface to HPSS. This particular example is for a Pget operation. The numbers indicate the main flow sequence. The PFTP client determines the parallelism required at its end (a function of the stripe width of the local file, local I/O architecture, and network connectivity); spawns client MVR threads or processes for each parallel connection that listen for connections; opens local and remote files; passes the local client MVR ports to the PFTP daemon; issues a retrieve command; receives the parallel data streams; and issues reads/writes to the local storage (which might be a local PFS). The parallel FTP daemon builds an HPSS IOD and calls the HPSS Client API readlist function. This results in parallel data transfer controlled by the HPSS and PFTP client MVRs.

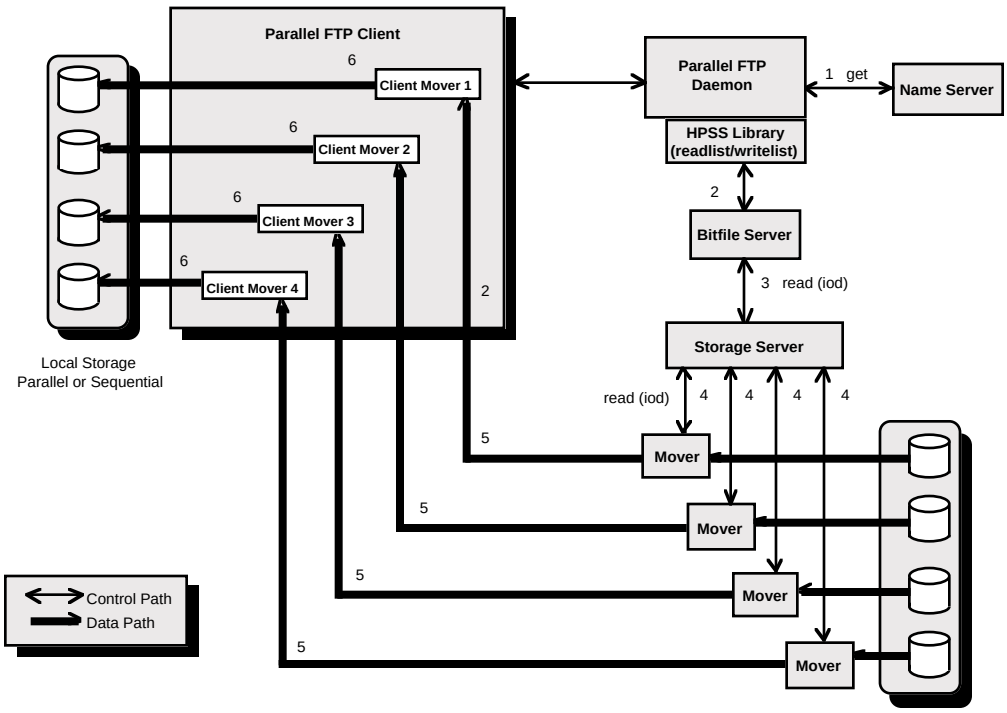


Figure 9. Parallel FTP transfer.

HPSS integration with a parallel system's local PFS

There is a spectrum of possible ways to integrate a parallel system's local PFS with a storage system such as HPSS. The characteristic that defines where an integration exists on this spectrum is the transparency, as seen by a user or application, of the name spaces and data locations of the local file system integrated with the storage system. At one end, which we will call the import/export end, an explicit command must be issued to move a file or partial file between a PFS and HPSS. At this end, two separate name spaces exist and multiple independent copies of the data exist. The use of FTP is an example of an integration mechanism at this end.

At the other end of the spectrum there is a single name space and data automatically caches to the local PFS from HPSS as it is accessed or automatically migrates to HPSS when not recently accessed and space must be freed for the PFS. Thus, the application logically sees a local PFS, which is a very large virtual store. The Unix VFS provides an example of a transparent integration mechanism for sequential access and transfers [31]. The DMIG has specified mechanisms that use a local file system's name space as seen by applications, but provides for transparent mechanisms to move whole files between local and remote systems [13,33]. These approaches do not, however, provide a uniform name space across multiple client systems. NFS and DFS servers integrated with storage provide additional transparency but do not support parallel I/O.

One would like to perform the integration such that high-speed parallel transfers exist between applications and PFS, and between PFS and HPSS, when appropriate connectivity supports parallel I/O. For example, NFS provides improvements in transparency but does not support parallel transfers to an application, although the HPSS NFS server does

support high-speed parallel transfers for caching and migration of large blocks. Similarly, the Andrew File System (AFS) [36] and DFS [40] support improvements in name space transparency over NFS, but again their protocols need to be modified to support parallel or even sequential high-speed, third-party transfers both to the application and between a DFS server and HPSS. An approach to interfacing AFS to storage systems is presented in Reference [38].

The HPSS project is either using or planning to use existing industry standard mechanisms such as NFS, VFS, DMIG, and DFS to provide increasing transparent integration of HPSS with PFS. It is also working with industry groups to improve the mechanisms and protocols to support transparent high-speed parallel integration of HPSS with PFS. An example is the PTP work described earlier and collaboration with the Scalable I/O Initiative [2]. However, many issues need resolution for full transparent parallel I/O integration at all system levels.

There are important classes of applications and configurations where it may be most appropriate for applications to bypass access to their local PFS and directly perform I/O to/from HPSS. Such environments would have relatively large I/O transfers, an HPSS configuration with appropriate connectivity between the parallel system and resources such as disks and tapes.

We are also working with vendors and users to utilize the most effective import/export mechanisms available with a given PFS to integrate with HPSS. An example of this type of integration is the use of HPSS to support parallel tape services on the IBM SPx. In the initial integration, the tapes are connected to SPx nodes. The goal is to support high-speed parallel import/export of data between the PFS controlled disks and the HPSS controlled tapes using the high-speed internal SPx switching network.

HPSS to SPx PFS parallel-to-parallel transfer

This particular example is for an export operation from PFS to HPSS. Figure 10 illustrates the control and data flow for an HPSS to IBM SPx Vesta or PIOFS parallel-to-parallel transfer [8,9]. The numbers on the figure indicate flow sequence. HPSS can run on SPx node(s) or a separate server. For each import or export request, an HPSS open API is issued, followed by a writelist API (for export) or readlist API (for import). The HPSS MVR has additional logic to issue PFS open and read (or write) APIs so only a single MVR is required. The second client MVR is implicitly in the PFS library and PFS. All other HPSS server processing is unaffected. Once the transfer has completed, the import/export daemon closes the HPSS file.

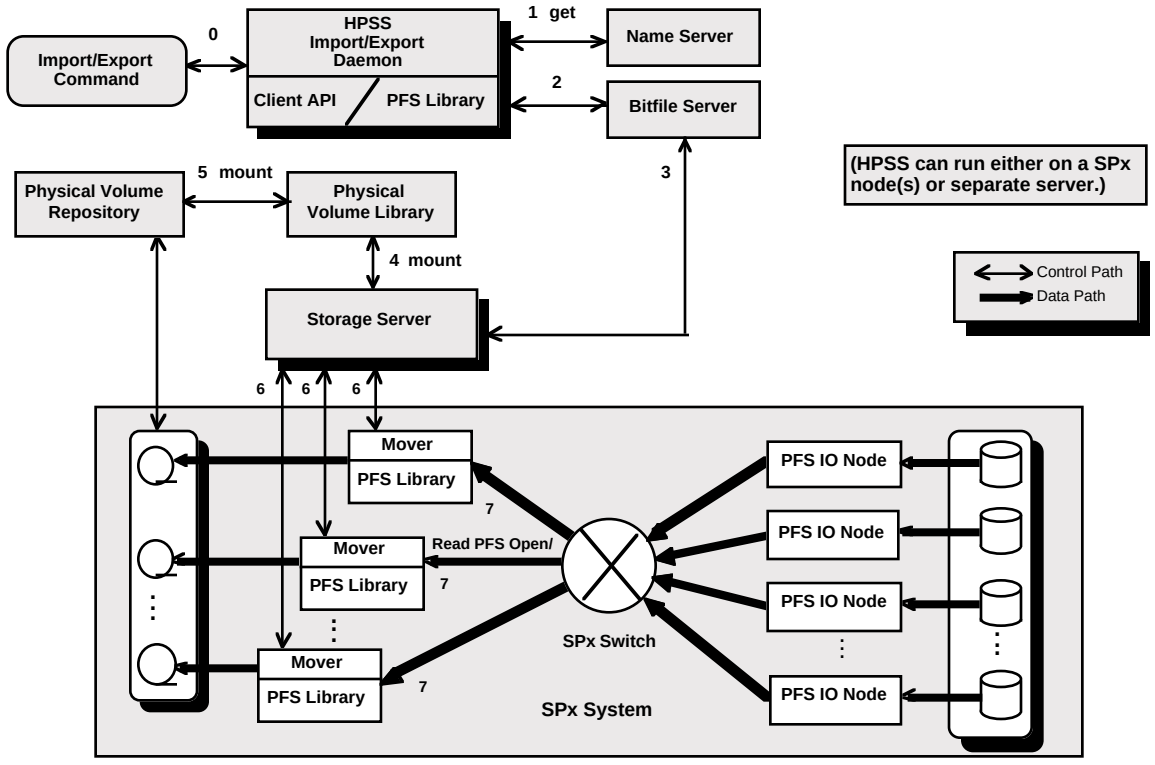


Figure 10. HPSS to SPx PFS parallel-to-parallel transfer.

Supercomputing ‘94 demonstration

HPSS parallel disk and tape I/O and parallel FTP were demonstrated at Supercomputing '94. The demonstration configuration (Figure 11) consisted of a four-node IBM SP2, RS 6000, and PsiTech frame buffer with Sony monitor as client systems. The available storage devices were four SCSI disks on the SP2 nodes, two HIPPI connected IBM 9570 RAID disks, two Ampex DST 600 D2 tape drives, four IBM 3480 tape drives, and an IBM NTP drive. The system was interconnected by a 32-node Network Systems Corp. HIPPI switch. We demonstrated data rates of 80 MB/s from the two 9570s to the SP2 (limited by the 40-MB/s SP2 channels), 26 MB/s from the two DST 600s to the frame buffer (we have also seen 39 MB/s from three Ampex drives to an RS 6000 at the NSL), and 15 MB/s from the four SCSI drives to the frame buffer.

(The RS 6000 was used as the HIPPI Controller for the Tape Systems.)

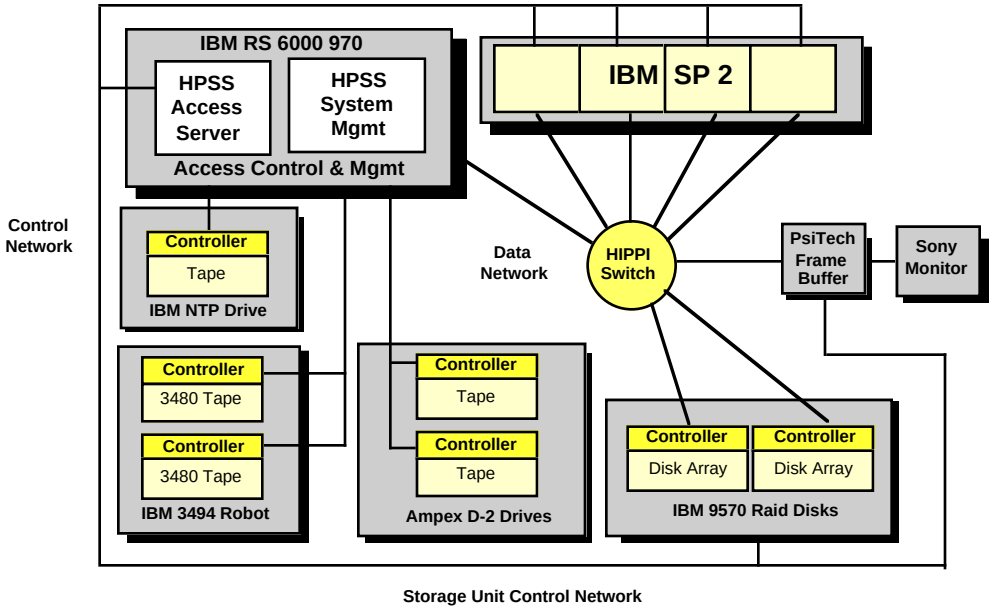


Figure 11. HPSS configuration at Supercomputing '94
(the RS 6000 was the HIPPI controller for the tape systems).

Summary

We have described the core parallel I/O mechanisms used in HPSS and its client interface services and given examples of their use. The basic I/O transfer model is based on the general PTP. The key HPSS components involved in the parallel I/O architecture are the SSs and the MVRs. The central data structure used for describing a parallel transfer is the IOD with its source and sink descriptor lists. A key architectural concept is successive expansion of the IOD to create the parallel transfer plan as it passes through the HPSS components.

Parallel I/O is a relatively new area of research, development, and practical application. There is much to be learned about parallel I/O and its use at various levels (application, library, language, and compiler, operating system, local PSF, and parallel and distributed storage systems). As we learn more about the total end-to-end parallel I/O environment and how to integrate the various levels into easy-to-use parallel programming models, APIs, and tools, we are confident that the modularity and parallel I/O architecture and mechanisms of HPSS have the flexibility to evolve and integrate with the evolving total parallel I/O environment. Based on our HPSS experience with our development and demonstration hardware, we have seen no limits to scalability introduced by the HPSS software. The performance limitations observed have been peripheral and host channel rates, number of available devices and their connectivity, and choices made for data layout such as stripe width and blocking factors at source and sink.

As this paper was written, HPSS Release 1 was in final integration testing, and design and implementation are underway for integration with the IBM SPx, Intel Paragon and Meiko CS 2 PSFs. Release 1 supports the architecture described here for parallel tape. Much of the coding for HPSS Release 2 has also been completed. Release 2 supports parallel disk, multiple hierarchies, and parallel migration and caching between storage devices at different hierarchi-

cal levels; general availability is planned for late 1995.

Acknowledgments

The credit for the parallel I/O design and implementation work described in this paper belongs to the members of the PTP design group and the HPSS Technical Team. Specifically, we want to acknowledge Peter Corbett, IBM TJ Watson; Kurt Everson and Rich Ruef, IBM; Erik DeBenedictis, Scalable Computing; Bill Nesheim, formally of Thinking Machines; and Larry Berdahl, LLNL, for their work on the PTP design—and Danny Cook, LANL; Dave Fisher, LLNL; and Kurt Everson and Rich Ruef of the IBM team for their HPSS parallel I/O work. We wish to acknowledge the many discussions and shared design, implementation, and operation experiences with our colleagues in the NSL collaboration, the IEEE Mass Storage Systems and Technology Technical Committee, the IEEE Storage System Standards Working Group, and in the storage community. Specifically, we wish to acknowledge the people on the HPSS Technical Committee and Development Teams. At the risk of leaving out a key colleague in this ever-growing collaboration, the authors acknowledge Dwight Barrus, Ling-Ling Chen, Ron Christman, Danny Cook, Lynn Kluegel, Tyce McLarty, Christina Mercier, and Bart Parliman from LANL; Larry Berdahl, Jim Daveler, Dave Fisher, Mark Gary, Steve Louis, Donna Mecozzi, Jim Minton, and Norm Samuelson from LLNL; Marty Barnaby, Rena Haynes, Hilary Jones, Sue Kelly, and Bill Rahe from SNL; Randy Burris, Dan Million, Daryl Steinert, Vicky White, and John Wingenbach from ORNL; Donald Creig Humes, Juliet Pao, Travis Priest and Tim Starrin from NASA LaRC; Andy Hanushevsky, Lenny Silver, and Andrew Wyatt from Cornell; and Paul Chang, Jeff Deutsch, Kurt Everson, Rich Ruef, Tracy Tran, Terry Tyler, and Benny Wilbanks from IBM Government Systems and its contractors.

We also want to acknowledge that the edited description of the PTP was based on Reference [1] edited by Larry Berdahl and most of the detailed examples and figures were taken from an early HPSS description written by Danny Teaff of IBM.

This work was, in part, performed by the Lawrence Livermore National Laboratory (contract no. W-7405-Eng-48), Los Alamos National Laboratory, Oak Ridge National Laboratory, and Sandia National Laboratories under auspices of the U.S. DOE Cooperative Research and Development Agreements; by Cornell Information Technology, NASA Langley Research Center, and NASA Lewis Research Center under auspices of the National Aeronautics and Space Agency; and by IBM Government Systems under Independent Research and Development and other internal funding.

References

1. Berdahl, L., ed., "Parallel Transport Protocol," draft proposal, available from Lawrence Livermore National Laboratory, Dec. 1994. The Parallel Transport draft is available from anonymous ftp svr4.nersc.gov, directory pub, file Pio-1-3-95.ps.
2. Bershad, B., et. al, "The Scalable I/O Initiative," White paper, available through the Concurrent Supercomputing Consortium, CalTech, Pasadena, Feb. 1993.
3. Buck, A. L., and R. A. Coyne, Jr., “Dynamic Hierarchies and Optimization in Distributed Storage System,” Digest of Papers, Eleventh IEEE Symposium on Mass Storage Systems, Oct. 7-10, 1991, IEEE Computer Society Press, pp. 85-91.
4. Carmen, T. H., and D. Kotz, "Integrating Theory and Practice in Parallel File Systems," Proc. DAGS/PC Symposium 1993, pp. 64-74.
5. Christensen, G. S., W. R. Franta, and W. A. Petersen, “Future Directions of High-speed Networks for Distributed Storage Environments,” Digest of Papers, Eleventh IEEE Symposium on Mass Storage Systems, Oct. 7-10, 1991, IEEE Computer Society Press, pp. 145-148.
6. Collins, B., J. Brewton, D. Cook, L. Jones, K. Kelly, L. Kluegel, D. Krantz, and C. Ramsey, “Los Alamos HPDS: High-Speed Data Transfer,” Proc. Twelfth IEEE Symposium on Mass Storage Systems, Monterey, April 1993.
7. Corbett, P. et al., "MPI-IO: a Parallel File I/OP Interface for MPI, version 3, available at http://lovelace.nas.nasa.gov/MPI-IO/mpi-io.html,"
8. Corbett, P. F., and D. G. Feitelson, "Design and Implementation of the Vesta Parallel File System," Proc. Scalable High Performance Computing Conference, 1994, pp. 63-70.
9. Corbett, P. F., D. G. Feitelson, S. J. Baylor, and J. Prost, "Parallel Access to Files in the Vesta File System," Proceeding of Supercomputing '93, IEEE Computer Society Press, Nov. 1993.
10. Coyne, R. A. and H. Hulen, “An Introduction to the Mass Storage System Reference Model, Version 5,” Proc. Twelfth IEEE Symposium

on Mass Storage Systems, Monterey, April 1993.

11. Coyne, R. A., H. Hulen, and R. W. Watson, “Storage Systems for National Information Assets,” Proc. Supercomputing 92, Minneapolis, Nov. 1992, pp. 626-633.
12. Coyne, R. A., H. Hulen, and R. W. Watson, "The High Performance Storage System," Proc. Supercomputing 93, Portland, Nov. 1993.
13. Data Management Interfaces Group, "Interface Specification," Version 2.0, Nov. 1994.
14. DeBenedictus, E., and S. Johnson, "Extending Unix for Scalable Computing," IEEE Computer, Nov. 1993.
15. del Rasario, J. M. and A. Choudhary, "High Performance I/O for Parallel Computers: Problems and Prospects" IEEE Computer, 27 (3): 59-68, March 1994.
16. Deutsch, J., and M. Gary, "Physcial Volume Library Deadlock Prevention in a Striped Media Environment," submitted to the Fourteenth IEEE Symposium on Mass Storage Systems, Monterey, Sept. 1995.
17. Dibble, P. C., "A Parallel Interleaved File System," Ph.D. Thesis, Univ. of Rochester, 1989.
18. Dietzen, Scott, Transarc Corporation, "Distributed Transaction Processing with Encina and the OSF/DCE," Sept. 1992, 22 pages.
20. Ghosh, J. and B. Agarwal, "Parallel I/O Subsystems for Distributed Memory Multicomputers," Proceedings of the Fifth International Parallel Processing Symposium, May 1991.
21. Golubchik, L., R. R. Muntz, and R. W. Watson. "Analysis of Striping Techniques in Robotic Storage Libraries." Proc Fourteenth IEEE Symposium on Mass Storage Systems, Monterey, Sept. 1995.
22. Grossman, R., H. Hulen, R. Coyne, T. Tyler, X. Qin, and W. Xu., "An Architecture for Scalable Digital Libraries" Proc Fourteenth IEEE Mass Storage Symposium, Sept. 1995.
23. Hartman, J. H., and J. K., Ousterhout, "The Zebra Striped Network File," Proc. Fourteenth ACM Symposium on Operating Systems Principles 1993, pp. 29-43.
24. Hyer, R., R. Ruef, and R. W. Watson, "High Performance Direct Network Data Transfers at the National Storage Laboratory," Proceedings of the Twelfth IEEE Symposium on Mass Storage, Monterey, IEEE Computer Society Press, April 1993.
25. IEEE Storage System Standards Working Group (SSSWG) (Project 1244), "Reference Model for Open Storage Systems Interconnection, Mass Storage Reference Model Version 5," Sept. 1994. Available from the IEEE SSSWG Technical Editor Richard Garrison, Martin Marietta (215) 532-6746
26. IEEE Storage System Standards Working Group, "Draft Mover Specification," in preparation.
27. "Information Technology -Open Systems Interconnection -Structure of Management Information -Part 4: Guidelines for the Definition of Management Objects," ISO/IEC 10165-4, 1991.
28. Internet Standards. The official Internet standards are defined by RFC's (TCP protocol suite). RFC 783; TCP standard defined. RFC 959; FTP protocol standard. RFC 1068; FTP use in third-party transfers. RFC 1094; NFS standard defined. RFC 1057; RPC standard defined.
29. ISO/IEC DIS 10040 Information Processing Systems - Open Systems Interconnection - Systems Management Overview, 1991.
30. Katz, R. H., “High Performance Network and Channel-Based Storage,” *Proceedings of the IEEE*, Vol. 80, No. 8, pp. 1238-1262, August 1992.
31. Kleiman, S. "V nodes: An Archtecturre for Multiple File System Types in SUN UNIX," Proc. of Summer USENIX Conference, 1986, Atlanta.
32. Lampson, B. W., "Atomic Transactions," in Distributed Systems -Architecture and Implementation, Berlin and New York: Springer-Verlag, 1981.
33. Lawthers, P., "Data Management Applications Programming Interface," Proc. Fourteenth IEEE Mass Storage Symposium, Sept. 1995.
34. Louis, S., and R. Burris, "Management Issues for High Performance Storage Systems," Proc Fourteenth IEEE Computer Society Mass Storage Systems Symposium, Sept. 1995
35. Louis, S., "Class of Service in the High Performance Storage System," Proc. IFIP International Conference on Open Distributed Processing Brisband, Australia, Feb. 1995.
36. Morris, J. H., et al., “Andrew: A Distributed Personal Computing Environment,” Comm. of the ACM, Vol. 29, No. 3, March 1986.
37. Nelson, M., et al., “The National Center for Atmospheric Research Mass Storage System,” Digest of Papers, Eighth IEEE Symposium on Mass Storage Systems, May 1987, pp. 12-20.

38. Nydict, D. et al., "An AFS-based Mass Storage System at the Pittsburgh Supercomputer Center" Digest of Papers Eleventh IEEE Symposium on Mass Storage Systems, Oct., 1991, pp. 117-122.
39. Open Software Foundation, Distributed Computing Environment Version 1.0 Documentation Set. Open Software Foundation, Cambridge, Mass. 1992.
40. Open Software Foundation, File Systems in a Distributed Computing Environment, White Paper, Open Software Foundation, Cambridge, MA, July 1991.
41. Peterson, D. G., and R. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)," Proc. SIGMOD Int. Conf. on Data Management, Chicago 1988, pp. 109-116.
42 Rullman, B., and D. Payne, "An Efficient File I/O Interface for Parallel Applications," Draft Intel Working Paper prepared for the Scalable I/O Workshop Frontiers '95.
43. Sandberg, R., et al., “Design and Implementation of the SUN Network File System,” Proc. USENIX Summer Conf., June 1989, pp. 119-130.
44. Sarawazi, S. "Database Systems for Efficient Access to Tertiary Memory" Proc Fourteenth IEEE Mass Storage Symposium, Sept. 1995.
45. Teaff, D., R. W. Watson, and R. A. Coyne, "The Architecture of the High Performance Storage System (HPSS)," Proceedings of the Goddard Conference on Mass Storage & Technologies, College Park, March 1995
46. Tolmie, D. E., “Local Area Gigabit Networking,” Digest of Papers, Eleventh IEEE Symposium on Mass Storage Systems, Oct. 7-10, 1991, IEEE Computer Society Press, pp. 11-16.
47. Tyler,T. W., IBM, and D. S. Fisher, LLNL, "Using Distributed OLTP Technology in a High Performance Storage System," Proc. Fourteenth IEEE Symposium on Mass Storage Systems, Monterey, Sept. 1995.
48. Watson, R. W., R. A. Coyne, "The National Storage Laboratory: Overview and Status," Proc. Thirteenth IEEE Symposium on Mass Storage Systems, Annecy France, June 12-15, 1994, pp. 39-43.
49. Witte, L. D., “Computer Networks and Distributed Systems,” IEEE Computer, Vol. 24, No. 9, Sept. 1991, pp. 67-77.

Technical Information Department • **Lawrence Livermore National Laboratory**
University of California • Livermore, California 94551